LENGUAJE de PROGRAMACION

Espacios de nombres (namespaces)

Un namespace es una zona separada donde se pueden declarar y definir objetos, funciones y cualquier identificador de tipo, clase, estructura, etc.; al que se asigna un nombre.

En general, los identificadores se declaran y definen fuera de cualquier namespace, en lo que se denomina el *espacio global*. Sin embargo, para evitar problemas de duplicación de nombres en grandes proyectos o cuando se usan bibliotecas externas, se pueden crear tantos namespaces como sean necesarios. Así, pueden existir objetos o funciones con el mismo nombre, declarados en diferentes ficheros fuente, siempre y cuando se declaren en distintos espacios de nombre. En esencia, un namespace define un alcance (scope).

- Definición de un namespace

```
namespace [<identificador>]                    // el nombre es opcional
{
   ...
   <declaraciones y definiciones>
   ...
}
```

- Los namespaces pueden anidarse.

Espacios de nombres (namespaces)

```
/***** Ejemplo: en el archivo de cabecera puntos.h se definen dos namespaces.
namespace espacio_2D
{
        struct punto { int x, y; };
        double distancia(punto);
}
 namespace espacio_3D
{
        struct punto { int x, y, z; };
        double distancia(punto);
}
fin del ejemplo *****/
```

- Directiva using: existen dos opciones para acceder a los datos/funciones de un namespace:

  a) using namespace <ns_nombre>;

  <variable> = valor;
  <funcion>(par1, par2, ...);

  b) <ns_nombre>::<variable> = valor;
  <ns_nombre>::<funcion>(par1, par2, ...);

- ✓ Los nombres introducidos con una directiva using cumplen con las reglas de ámbito: el nombre es visible desde el punto en que se incluye la directiva hasta el final de su alcance

Espacios de nombres (namespaces)

```
/***** Ejemplo
#include "puntos.h"
 using namespace espacio_2D;
 int main()
{
        …
        punto p1;                       // define la variable p1 de tipo espacio_2D::punto
        espacio_3D::punto p2;           // define la variable p1 de tipo espacio_3D::punto
        …
        d1 = distancia(p1);             // invoca a la función definida en espacio_2D
        d2 = espacio_3D::distancia(p2); // invoca a la función definida en espacio_3D
        …
}
fin del ejemplo *****/
```

- ✓ La directiva using se puede utilizar para referirse a un ítem particular dentro del namespace. En este caso, el resto de las declaraciones no son accesibles si no se utiliza el operador ::.

```
using espacio_2D::punto;                // sólo es accesible punto de espacio_2D
{
      …
      punto p1;                         // define la variable p1 de tipo espacio_2D::punto
      d = distancia(p1);                // error, el compilador no accede a distancia()
      d = espacio_2D::distancia(p1);    // correcto (uso operador especificador de ámbito)
      …
}
```

Espacios de nombres (namespaces)

En C++ todas las funciones estándar se incluyen en el espacio de nombres **std**, por lo que es aconsejable incluir la directiva **using namespace std** al principio de los programas. Las variables y funciones a las que se tiene acceso en este namespace se encuentran definidas en distintos archivos de cabecera que se caracterizan por no incluir el sufijo **.h**. Por lo tanto, para utilizar las librerías estándar en C++ se debe utilizar #include<libname> en vez de #include<libename.h>. Algunos de los nuevos nombres de los archivos de cabecera son:

C	**C++**
#include <stdlib.h>	#include <cstdlib>
#include <math.h>	#include <cmath>
#include <stdio.h>	#include <cstdio>
#include <iostream.h>	#include <iostream>

- Alias: se puede crear un nombre alternativo para un namespace para distinguirlo con mayor facilidad ó diferenciarlo de otro con nombre parecido o complicado (demasiado largo).

  namespace <alias_de_espacio> = <nombre_de_espacio>;

- Namespace sin nombre: permite crear identificadores visibles sólo en determinadas zonas, las variables serán accesibles sólo en el fichero fuente donde se define el namespace.

Variables de tipo referencia

En C++ se presenta una nueva forma de pasar argumentos *por referencia* a una función: tipo de dato **reference**. Una referencia a una variable es un "alias" de la misma (misma posición de memoria). Para declararla se le antepone al nombre el caracter **&** (no confundir con el operador dirección) y *debe ser inicializada a otra variable o a un valor numérico*:

```
int& iref = i;          // declaración de referencia válida  (suponiendo que i es tipo int)
int& jref;              // declaración de referencia no válida (no se inicializó)
```

Una vez que iref se declaró como alias de i no se puede declarar como alias de otra variable (diferencia con punteros). Los arrays no pueden ser declarados como variables referencia. El principal uso de las variales referencia es como valor de retorno de una función o como argumentos de la misma. Es mucho más rápido pasar un puntero a un alias que una copia del valor, principalmente con estructuras complejas.

Se pueden pasar argumentos por referencia anteponiendo el carácter (&) tanto en el prototipo como en el encabezamiento de la definición. Dentro de la función no se utiliza el operador *indirección* (*). En la llamada a la función los argumentos se ponen sin operador.

Punteros, referencias y la mayoría de tipos básicos ocupan lo mismo en el **stack**, por lo que no se obtiene ninguna ventaja al pasar un tipo básico como referencia constante.

Ej. 01: Var refer

Expansión *inline*

Es común en POO realizar muchas llamadas a funciones que contienen pocas sentencias, o incluso una sola (lectura y asignación de valores a variables). Cada llamada y retorno de una función tiene un costo computacional, debe reservarse memoria para los argumentos que deben ser copiados. Para funciones sencillas, el tiempo de transmisión de datos puede ser mayor que el necesario para realizar los cálculos. Como solución se realiza una expansión ***inline*** de la función, que sustituye la llamada por el código de la misma, reduciendo el tiempo de ejecución de un programa. La implementación se puede realizar de dos maneras:

✓ definición de la función: ***inline*** *double* <nombre_funcion> (*double* par) { *return* uno; }

Puede ser ignorada por el compilador en el caso de que la función en cuestión sea tan larga o tan complicada que su expansión resulte desaconsejable. La definición debe hacerse en los archivos header (*.h), y no en los fuente (*.cpp) para que sea accesible desde otros archivos.

✓ consiste en colocar la **definición** completa de la misma en la **declaración** de la función:

```
class <nombre_clase>
{   …
    double <funcion_inline>(param)    // se pueden incluir argumentos por defecto
    …                                 // no es necesaria la palabra inline (menor legibilidad)
}
```

Entrada / salida (*streams*)

Las operaciones de entrada y salida no forman parte de C y C++ (son bibliotecas externas). En C++ se dispone de varias clases para el manejo de **streams** (flujo de datos entre una fuente y un destinatario): **streambuf** (manejo de buffers entre la memoria y los dispositivos físicos), **ios** (entrada y salida, incluye un objeto *streambuf* en su definición), **istream**, (derivada de *ios*, especializada en entradas), **ostream** (derivada de *ios*, especializada en salidas), **iostream** (derivada de *istream* y *ostream*, encapsula las funciones de entrada y salida por teclado y pantalla) y **fstream** (entrada y salida con ficheros). Dejar fuera del lenguaje todas las operaciones de entrada y salida tiene varias ventajas:

- ✓ Independencia de la plataforma: cada compilador dispone de diferentes versiones de cada biblioteca de acuerdo a la plataforma donde se instala. Se mantienen inalterables los parámetros de entrada a cada función y sus valores de retorno.
- ✓ Encapsulación: para el programa todos los dispositivos son de entrada y salida se tratan del mismo modo, es indiferente usar la pantalla, el teclado o ficheros.
- ✓ Buffering: el acceso a dispositivos físicos es lento, en comparación con el acceso a memoria. Las operaciones de lectura y escritura se agrupan, haciéndolas en memoria, y las operaciones físicas se hacen por grupos o bloques, lo cual ahorra mucho tiempo.

Entrada / salida (*streams*)

El archivo de cabecera iostream incluye el namespace **std** que define varios streams estándar: **cin** (entrada de datos, en general teclado), **cout** (salida de datos, en general pantalla), **cerr** (mensajes de error por pantalla). Al utilizar estos métodos se ajustan automáticamente las entradas/salidas al *tipo* de dato. Si en C se cambia el *tipo* de la variable, hay que modificar los strings de formatos (%....) en *printf* o *scanf*.

Para utilizar los objetos cout y cin se tienen dos opciones (luego del #include <iostream>):

a) **using namespace std**;

```
....
cout << "Mensaje de salida" << endl;
...
```

b) ....

```
std::cout << "Mensaje de salida" << std::endl;
...
```

El uso del *namespace std* evita tener que utilizar el operador de resolución en todas las referencias a *cout* (*cin*).

Objeto cout

Cuenta con el operador << (de inserción) que se encuentra sobrecargado para ser utilizado con todos los tipos estándar: *char*, *char* *, *void* *, *int*, *long*, *short*, *bool*, *double* y *float*. El operador devuelve una referencia de un objeto *ostream* de modo que puede concatenarse para formar expresiones del tipo (evalúa de izquierda a derecha):

```
cout << "El valor de la variable es: " << variable << "\n" << endl;
```

- **endl**: vacía el buffer de salida, la siguiente salida se imprime en una nueva línea. Se puede invocar como cout.endl() ó cout << endl.
- **flush**: vacía el buffer de salida. Se puede usar como cout.flush() ó cout << flush.
- **width**: devuelve [num = width()] o fija [width(num)] el ancho de la salida.
- **fill**: devuelve [num = fill()] o fija [fill(car)] el caracter que rellena el ancho de la salida.
- **precision**: devuelve [num = precision()] o fija [precision(num)] la cantidad de dígitos totales que tendrá el número.
- **flags**: devuelve [numl = flags()] o fija [flags(numl)] los *flags* del formato de salida.
- **put**(*char*): imprime un caracter.
- **write**(*char* *cad, *int* n): imprime n caracteres de cad.

Objeto cout

El formato de la salida se puede modificar mediante el uso de flags y manipuladores (funciones), que se encuentran definidos en el archivo de cabecera **iomanip** (el **.h** es opcional).

- Manipuladores con parámetros:
  - ✓ **setw**(*width*): ajusta el ancho de impresión de la siguiente salida.
  - ✓ **setfill**(*car*): completa con '*car*' el número hasta el ancho fijado. Afecta a todas las salidas posteriores. Para anularlo usar **cout.setfill**(' ') (llena con espacios).
  - ✓ **setprecision**(*num*) imprime *num* dígitos en **total**. Si se necesita fijar la cantidad de dígitos **decimales** se debe utilizar primero **setiosflags**(**ios::fixed**). El ajuste se mantiene para las salidas posteriores. Tener en cuenta que se produce un redondeo al mostrar el valor.
  - ✓ **setiosflags** / **resetiosflags** (*flag*): activa o desactiva el *flag* de formato de salida (tabla).
  - ✓ **setbase**(*base*): convierte los números a la *base* especificada.
- Manipuladores sin parámetro: **dec**, **oct**, y **hex**. En general ofrecen la misma funcionalidad que los *flags*. La diferencia es que los cambios son permanentes, afectan a todas las salidas hasta que se vuelva a modificar el formato afectado.

Objeto cin

Cuenta con el el operador >> (de extracción) que se encuentra sobrecargado para ingresar el valor de una variable de todos los tipos estándar: *char&*, *char* *, *int&*, *long&*, *short&*, *double&* y *float&*. El operador devuelve una referencia de un objeto *istream* de modo que puede concatenarse para formar expresiones del tipo (evalúa de izquierda a derecha):

```
cin >> var1 >> var2;
```

Cuando se usa >> para cadenas, la lectura se interrumpe al encontrar un carácter '\0', ' ' ó '\n'. Además se debe tener en cuenta que cin no comprueba el desbordamiento del espacio disponible para el almacenamiento de la cadena (posibilidad de invadir memoria).

- **width**: permite fijar la cantidad de caracteres máxima que se ingresa a una cadena.

```
char cadena[10];
cin.width(sizeof(cadena));
cin >> cadena;
```

- **ws**: provoca que se ignoren los espacios iniciales en una entrada de tipo string.
- **flags**: devuelve [numl = flags()] o fija [flags(numl)] los *flags* del formato de entrada.
- **read**(*char* *cad, *int* n): lee n caracteres y los almacena a partir de la dirección de cad.

Objeto cin

- **get**: tiene tres formatos:
  - ✓ *int* get(); // lee un caracter y lo devuelve como valor de retorno (forma obsoleta).
  - ✓ *istream*& get(*char*& c); // lee un caracter, permite asociar → cin.get(a).get(b).get(c)
  - ✓ *istream*& get(*char** ptr, *int* len, // lee caracteres hasta un máximo de 'len' o hasta que *char* delim = '\n'); // se encuentre el carácter delimitador ('\n' por defecto)
- **getline**: es equivalente a la versión con tres parámetros de get(), con la diferencia que el carácter delimitador también se lee (con get no).
- **ignore**(*int* n=1, *int* delim = EOF): ignora los caracteres que están pendientes de ser leídos, por ejemplo cuando se fijó width y el usuario ingresa una cadena más larga.
- **peek**: obtiene el siguiente caracter del buffer de entrada pero no lo retira.
- **putback**(*char*): coloca un caracter en el buffer de entrada.

Ej. 02: Cin Cout

Gestión dinámica de memoria

Se utilizan los operadores **new** y **delete**. Una variable creada con **new** dentro de cualquier bloque, perdura hasta que es explícitamente borrada con **delete** (puede ser manipulada por instrucciones de otros bloques). El operador **new** permite crear variables de cualquier tipo (no es necesario hacer un **cast**), incluso de tipos definidos por el usuario. Al utilizar el operador **delete** se borra la memoria que ocupaba la variable y se reutiliza. La sentencia:

MiClase *ptr = **new** *MiClase*;

a) Crea un puntero ptr capaz de contener la dirección de un objeto de la clase *MiClase*.

b) Mediante el operador new se reserva memoria para un objeto del tipo *MiClase*.

c) Se llama, de modo transparente al usuario, a un *constructor* de *MiClase*, para inicializar las variables miembro.

- Para crear arreglos de objetos:

MiClase *ptr = **new** *MiClase*[tamaño];

Gestión dinámica de memoria

La diferencia fundamental entre new/delete y malloc()/free() es que los primeros crean y destruyen **objetos,** mientras que los segundos se limitan a reservar y liberar zonas de memoria. Además, new puede ser sobrecargado como cualquier otro operador.

Al utilizar delete (ó free), se libera la zona de memoria a la que apunta, *pero sin borrar el propio puntero*. Si se ordena *liberar dos veces* lo apuntado las consecuencias son imprevisibles, por lo que es importante evitar este tipo de errores.

En el caso de que se desee liberar la memoria ocupada por un vector creado mediante reserva dinámica de memoria debe emplearse una instrucción del tipo:

```
delete [] ptr;              // ptr es el puntero asignado por new
```

Ej. 03: New Delete

Constructores

Son funciones miembro públicas que tienen el mismo nombre que la clase y no tienen valor de retorno, ni siquiera *void*. Se llaman de modo automático cada vez que se crea un nuevo objeto y su misión es iniciar correctamente las variables miembro. Se pueden definir *varios constructores*, dependiendo de la cantidad de parámetros por defecto que se utilicen.

```
<nombre_clase> :: <nombre_clase>(parámetros)   // clase y constructor con igual nombre
{
    <nombre_variable1> = parámetro1;              // variables miembro privadas
    <nombre_variable2> = parámetro2;
}
```

- C++ permite *inicializar* variables miembro fuera del cuerpo del constructor, de la forma:

```
<nombre_clase> :: <nombre_clase>(parámetros) :    // ':' después del constructor
    <nombre_variable1> (valor1) ,                 // nombre_variable1 = valor1
    <nombre_variable2> (valor2)                   // nombre_variable2 = valor2
     { ... ;  }                          // en este caso el cuerpo del constructor está vacío
```

Los *inicializadores* se colocan, tras el carácter '**:**' separados por '**,**' antes del cuerpo del constructor. Forma: nombre de la variable miembro y entre paréntesis el valor asignado.

Constructores

- La llamada al constructor se puede hacer explícitamente en la forma:

  nombre_clase <nuevo_objeto> = *nombre_clase*(parámetros) // valores de las variables

  o bien, de una forma implícita, más abreviada:

  nombre_clase <nuevo_objeto>(parámetros) // sin utilizar el signo '='

- *por defecto*: o no tiene argumentos, o si los tiene, todos tienen asignados un valor por defecto en la declaración. En ambos casos puede ser llamado sin pasarle ningún parámetro.

- *de oficio*: cada vez que se crea un objeto de una clase, C++ **obliga a inicializar** sus variables miembro llamando a un constructor. En el caso de que el programador **no defina ningún constructor**, el compilador define automáticamente un constructor *por defecto* sin argumentos; que inicializa todas las variables miembro a cero. Todo constructor de oficio es constructor por defecto, pero el usuario puede definir constructores por defecto que no son de oficio.

Constructores

- *de copia* (copy constructor): inicializa un nuevo objeto copiándolo desde otro objeto de la misma clase. Tiene un único argumento que es una *referencia constante al objeto fuente*. El compilador proporciona un constructor de copia de oficio que realiza una copia bit a bit de las variables miembro (no deseado con punteros = no copia los contenidos de la memoria sino las direcciones). Al trabajar con punteros el constructor de copia debe realizarlo el programador para gestionar espacios de memoria distintos y luego copiar sus contenidos.

```
tipo_clase objeto1(valor1, valor2);        // constructor con argumentos
tipo_clase objeto2 = objeto1;              // signo '=' sobrecargado
tipo_clase objeto3(objeto1);               // constructor de copia
```

Existen dos casos muy importantes en los que hay que utilizar copias del objeto:

✓ Cuando a una función se le pasan objetos como *argumentos por valor.*

✓ Cuando una función tiene un *objeto como valor de retorno*.

Destructores

El **destructor** es llamado automáticamente por el sistema operativo cuando el objeto va a dejar de existir, al finalizar un bloque si es local o al finalizar el programa si es global. Los objetos creados con el operador **new** perduran hasta que se invoque el operador **delete**, por lo que la liberación de recursos es responsabilidad del programador. El destructor siempre es **único** (no puede estar sobrecargado), **nunca tiene argumentos ni valor de retorno**. Su nombre es el mismo de la clase precedido por el carácter '~' (Alt+126). Si no se define, el compilador proporciona un destructor de oficio, que es adecuado excepto para liberar memoria dinámica (punteros).

```
<nombre_clase> :: ~<nombre_clase>()      // clase y destructor con igual nombre + '~'
{
      delete [ ] ptr;                    // libera la memoria apuntada por ptr
      ........;                          // sentencias necesarias para liberar otros recursos
}
```

Variables miembro *static*

Se puede necesitar que una variable miembro sea común para todos los objetos de la clase, de modo que todos compartan el mismo valor (interés ofrecido por un banco es el mismo para todas las cuentas). Para este caso C++ permite declarar una variable miembro como **static**, sea public, protected o private.

- Sólo existe una copia de cada una de las variables miembro static. Todos los objetos declarados de esa clase *hacen referencia a la misma a la misma posición de memoria*.
- Las variables static de una clase existen aunque no se haya declarado ningún objeto. La variable debe declararse dentro de la definición de la clase y después debe definirse (inicializarse) como global en el programa que utilizará la clase.
- Para referirse a una variable static se puede utilizar el nombre de un objeto y el operador punto (**.**) aunque es confuso (se está haciendo referencia a una variable común a todos los objetos mediante el nombre de uno sólo de ellos). Es mejor utilizar el nombre de la clase y el scope resolution operator (**::**): *nombre_clase*::*variable_static*.
- Son de gran utilidad para llevar un contador del número de objetos creados de una clase. El contador se incrementa con cada constructor invocado y se disminuye con el destructor.

Funciones miembro *static*

- Son funciones genéricas que no actúan sobre ningún objeto concreto de la clase. Permite acceder a las variables static privadas.

- No pueden acceder a variables miembro que no sean static.

- No pueden utilizar el puntero this (éste hace referencia a un objeto concreto).

- Pueden ser llamadas indiferentemente con el operador '**.**' ó el operador '**::**'.

Ej. 05: Var Static

Modificador *volatile*

- Variables y argumentos cuyos valores se pueden cambiar fuera del programa:

  volatile tipo <nom_variable> [= valor]; // variable volatile

  <nom_funcion> (*volatile tipo* <nom_variable>) // argumento volatile

- Aplicado a métodos de clase, la función debe ser invocada por objetos también volatile:

  tipo <nom_funcion> (*tipo* <nom_variable>) *volatile*; // función volatile

- Aplicado a instancias de la clase (objetos): se agrega una característica de seguridad. Si un objeto volatile invoca a una función miembro no-volatile, el compilador produce un error.

  volatile MyClass <nombre_obj_vol>; // instancia de MyClass (objeto)

La clase declarada puede tener dos versiones de una misma función (una volatile y la otra no). No se trata de un caso de polimorfismo, porque ambas tienen exactamente la misma definición (cantidad y tipo de parámetros). Cuál de las dos se utilizará en cada invocación de un objeto depende de que el propio objeto sea declarado volatile o no.

Modificador *mutable*

Se utiliza para indicar que un miembro particular de una clase (estructura) puede ser alterado, aún cuando la misma se ha declarado const. Es útil cuando se necesitan que los atributos de la clase no cambien salvo uno en particular; por ejemplo en un structura que tiene datos personales se puede hace que sólo se pueda variar la propiedad *password*.

```
struct <datos>
{
   char nombre[30];    /* ……*/          // otros datos personales
   mutable int accesos;                 // cantidad de veces que accede a la aplicación
};
const datos cargo = {"Pablo Picapidera",  "gerente",  /*otros datos*/,  0};
strcpy(cargo.nombre, "Pedro Marmol");     // no permitido, es const
cargo.accesos++;                          // permitido
```

Modificador *const* con función

Colocado después de la declaración de una función significa que no se le permite a la misma cambiar ninguno de los miembros de la clase (excepto que se haya modificado con mutable). Por lo tanto sólo tiene sentido su aplicación en funciones miembro.

Clase *string*

- Declaración y asignación: a un objeto de tipo string, se le pueden asignar otros objetos del mismo tipo, cadenas de caracteres (entre comillas dobles) e incluso un único carácter.

```
#include <string>

int main()
{
   string cad_1("Primer string");          // forma 1 de inicializar
   string cad_2 = "Segundo string";        // forma 2 de inicializar
   cad_2 = cad_1;                          // copia un string en otro
   cad_1 = 'H';                            // asigna un caracter a un string
}
```

- Acceso a un carácter: a través del método **at**(*pos*) ó, considerando que el string es un array, utilizando <nombre_str>[*pos*].

```
string strTexto("Mensaje");
char c;
c = strTexto.at(1);                        // retorna 'e'
c = strTexto[2];                           // retorna 'n'
```

- Comparaciones en forma alfabética: se realiza mediante los operadores lógicos ==, !=, <=, >=, <, >. Se distingue entre letras mayúsculas y minúsculas.

Clase *string*

- Concatenación: mediante el operador '+' se concatenan dos o más cadenas de caracteres, de forma equivalente a la función *strcat*() de C.

- Búsqueda de subcadenas o caracteres: el método **find**(*str*, *off*) retorna la posición a partir de la cual se encuentra *str* (puede ser un caracter ó una cadena) dentro del objeto string. El parámetro *off* fija la posición a partir de la cual se realiza la búsqueda. Retorna string::npos (-1, equivalente a NULL en el tipo string) si no se encuentra.

  El método **find_first_of**("caj", *off*) retorna la posición de la primera aparición de alguno de los caracteres de *str* ('c', 'a' ó 'j').

  El método **find_last_of**("caj", *off*) retorna la posición de la última aparición de alguno de los caracteres de *str* ('c', 'a' ó 'j'), recorriendo string desde el final al comienzo (reverse). Puede resultar útil para buscar el último '/' en un path.

- Obtener una subcadena: el método **substr**(*pos*, *num*) permite obtener una subcadena a partir del índice *pos* de la principal. Con el parámetro *num* se puede especificar la longitud de la subcadena, si no se utiliza se retornan los caracteres restantes del objeto string.

Clase *string*

- Tamaño de un string: existen dos métodos que permiten conocer la longitud de string: **length**() y **size**(). Ambos son equivalentes y se invocan sin parámetros.

  El método **resize**(*new_size*, *car*), permite cambiar la longitud de la cadena (acortarla o hacerla más larga). El parámetro *car*, cuando se especifica, permite inicializar los nuevos elementos con ese valor.

- Método **c_str**(): retorna el arreglo de caracteres asociados al objeto string, incluyendo el '\0' final. Si no se necesita el caracter de fin de línea se debe utilizar el método **data**().

- Método **empty**(): retorna un booleano que permite conocer si el string está vacío (TRUE). Es útil cuando se quiere validar si algún campo de texto de la interfaz con el usuario contiene o no datos.

- Método **swap**(*str_from*): permite intercambiar las cadenas de caracteres entre el objeto que invoca al método y el string *str_from*.